A Small Part of Common Lisp

Michael A. Covington
Institute for Artificial Intelligence
The University of Georgia

Program layout

A Lisp program is a text file containing symbolic expressions (S-expressions). By typing the expression

```
(load "c:\\temp\\myfile.lisp")
```

or the like, you tell the Lisp system to read the file and evaluate all the expressions on it (but not print their values).

Indentation is entirely up to you. Any expression can span more than one line; it is defined by its parentheses. A semicolon makes the rest of the line a comment.

Upper and lower case in symbols are not distinguished. Thus `X` and `x` are the same symbol. Hyphens and asterisks are permitted in symbols (`like-this`, `*like-this*`).

Types of S-expressions

Numbers

Integers: `2    2342342342   -234`

Rationals: `2/3    11/5   -2/7`

Floating-point numbers: `2.0     2.3   -0.999988    1.4E6` ($= 1.4 \times 10^6$)

Symbols: `x   y   z   first-example   *or-what*`

The empty list: `nil   ()`

Lists (non-empty): `(a b c) (a list (within a) list)`
Elements can be any type of S-expression.

Strings (of characters): `"This is a string"`

Lisp also has notations for other abstract objects. For example, `#'abc` denotes the function whose name is `abc`.

Evaluation

Numbers, strings, and `nil` evaluate as themselves.

Symbols are treated as variables and evaluate to whatever value is bound to them. Evaluating a symbol that has not been bound is an error. (You can't have uninitialized variables in Lisp.)

The symbol `t` evaluates to itself. It stands for "true."

Lists evaluate as calls to functions or special forms. For example, the list `(+ 2 3)` calls the function named `+` with arguments `2` and `3`. The list `(X)` calls the function named `X` (if there is one) with no arguments.

Functions evaluate all their arguments before applying the function. For example:

```
(+ (+ 2 3) (+ 9 8))
⇓
(+ 5 17)
⇓
22
```

Special forms do not evaluate all their arguments. The definition of the special form tells you which arguments, if any, get evaluated.

What we mean by "gives" or "returns"

If we say that a function "**gives**" or "**returns**" or "**evaluates to**" or "**evaluates as**" 234, that means its value is 234.

In this document, evaluation is symbolized by ⇒. Thus, "`(* 6 7)` ⇒ `42`" means that the value of `(* 6 7)` is `42`.

The purpose of functions is to compute values, not to print things. The computer may or may not print out the value of a function, depending on the circumstances.

Consider for example the expression `(+ 5 (* 3 4))`. Clearly the purpose of `(* 3 4)` is to *return* (*give*) the value 12 for use in further computation. A function that *prints out* 12 but does not return 12 as its value would be useless there.

Arithmetic

Types: The result of an integer division is an integer or rational number. Otherwise, arithmetic follows the rule that if you mix integers with rationals, you get a rational result, and if you mix any kind of number with floating-point numbers, you get a floating-point result.

`(+` *number number number...* `)` (Function)

Gives the sum of the numbers.

`(-` *number number number...* `)` (Function)

Gives the result of subtracting all the subsequent numbers from the first one.

`(*` *number number number. . .* `)` (Function)

Gives the product of the numbers.

`(/` *number1 number2* `)` (Function)

Gives $\frac{number1}{number2}$. If both numbers are integers, result is either integer or rational.

Control of evaluation

`'` (quote) (Special form)

Prevents evaluation of its argument.

`'(a b c)` ⇒ `(a b c)`

`` ` `` (backquote) (Special form)

Prevents evaluation of its argument, but allows expressions within it to be evaluated if they are marked with commas.

`` `(a ,(+ 2 3) c) `` ⇒ `(a 5 c)`

`(eval` *expression* `)` (Function)

Causes *expression* to be evaluated an extra time.

`(eval '(+ 2 3))` ⇒ `5`

This makes it possible to evaluate expressions you have constructed by computation.

Example:
`(cons '+ '(2 3))` ⇒ `(+ 2 3)`, so...
`(eval (cons '+ '(2 3)))` ⇒ `5`

`(setf` *symbol expression* `)` (Special form)

Gives the value of *expression*.

As a side effect, *symbol* is bound to the value of *expression*.

If *symbol* is not already a variable, it is created and is global.

`(setf x (+ 2 3))` ⇒ `5` and binds `x` to the number `5`.
`(setf x '(+ 2 3))` ⇒ `(+ 2 3)` and binds `x` to the list `(+ 2 3)`.

List manipulation

`(cons` *item list* `)` (Function)

Gives the result of putting *item* onto the beginning of *list*.

`(cons 'a '(b c))` ⇒ `(a b c)`

If *list* is not a list, the result is an "improper list" or "dotted pair":
`(cons 'a 'b)` ⇒ `(a . b)`

`(first` *list* `)` or `(car` *list* `)` (Function)

Gives the first element of the list.

`(first '(a b))` ⇒ `a`

`(rest` *list* `)` or `(cdr` *list* `)` (Function)

Gives the list of all elements of *list* except the first.

`(rest '(a b c d))` ⇒ `(b c d)`
`(rest '(a))` ⇒ `nil`

`(append` *list list list...* `)` (Function)

Joins two or more lists, giving a list of all their elements in order.

`(append '(a b c) '(d e f))` ⇒ `(a b c d e f)`
`(append '(a b) '(c d) '(e f))` ⇒ `(a b c d e f)`
`(append '(a b (c d)) '(e f))` ⇒ `(a b (c d) e f)`

Defining functions

`(defun` *name* `(` *argument argument argument...* `)` (Special form)
    expression
    expression
    expression
    ⋮
`)`

Evaluates to *name*, and as a side effect, defines (compiles) a function named *name* which will execute by substituting values for all the *argument*s and then evaluating all the *expression*s and returning the value of the last one.

Often there is only one *expression,* which is complex.

Example:

```
(defun double (x)
    (+ x x)
)
```

⇒ `double` and defines the function named `double`.

When this has been done, `(double 33)` ⇒ `66`.

Note: A program can add code to itself by creating a `defun` through computation and then evaluating it.

Comparisons and logic

Lisp uses `nil` (the empty list) to mean “false.” The special symbol `t` evaluates to itself and is usually used to represent “true,” although in fact any non-nil value counts as true. (The number 0 is non-nil.)

`(symbolp` *expression* `)` (Function)
`(numberp` *expression* `)` (Function)
`(listp` *expression* `)` (Function)

Gives `t` if its argument is of the type specified, and `nil` otherwise.

Note that `listp` is true of `nil` as well as non-empty lists.

```
(symbolp 'a) ⇒ t
(symbolp '(a b)) ⇒ nil
```

```
(listp 'a) ⇒ nil
(listp '(a b)) ⇒ t
(listp (cons 'a '(b))) ⇒ t
```

In these function names, *p* stands for *predicate.*

`(null` *expression* `)` or `(not` *expression* `)` (Function)

Gives `t` if its argument is `nil`, and `nil` otherwise.

`(>` *expression expression* `)` (Function)
`(<` *expression expression* `)` (Function)

These functions compare numbers in the expected way.

```
(> (+ 2 3) 4) ⇒ t
(> (+ 2 1) 4) ⇒ nil
```

`(equal` *expression expression* `)` (Function)

Gives `t` if its arguments are equal (having the same values arranged in the same structure, whether or not stored at the same memory location), `nil` otherwise.

```
    (equal '(a b)  (cons 'a '(b))) ⇒ t
(equal 2 (+ 1 1)) ⇒ t
(equal '(a b) '(a b c)) ⇒ nil
```

`(and` *expression expression expression...* `)` (Special form)

Gives the value of the last *expression* if all are non-nil; otherwise gives `nil`. Thus, you get a non-nil value only if all the expressions evaluate to non-nil.

Procedurally, `and` starts evaluating the expressions from left to right until it gets a `nil` value or until it runs out of expressions; then it gives you the value of the last one that it evaluated. Expressions after the first nil one are not evaluated.

```
(and (listp '(a b)) (symbolp 'a)) ⇒ t
```

`(or` *expression expression expression...* `)` (Special form)

Gives the value of the first *expression* that has a non-nil value, or `nil` if all are nil.

Procedurally, `or` starts with the first expression, and evaluates expressions one by one until it gets one that is not nil, and then stops. Thus, you get a non-nil value if at least one of the expressions evaluates to non-nil, and the expressions after the first non-nil one are not evaluated.

```
(or (listp '(a b)) (symbolp '(a b))) ⇒ t
```

Decision-making

`(if` *expression1 expression2 expression3* `)` (Special form)

Evaluates *expression1*. If the result is non-nil, proceeds to evaluate *expression2*; otherwise evaluates *expression3*. Value of the whole thing is the value of the last of the arguments that was evaluated.

Expression3 is optional.

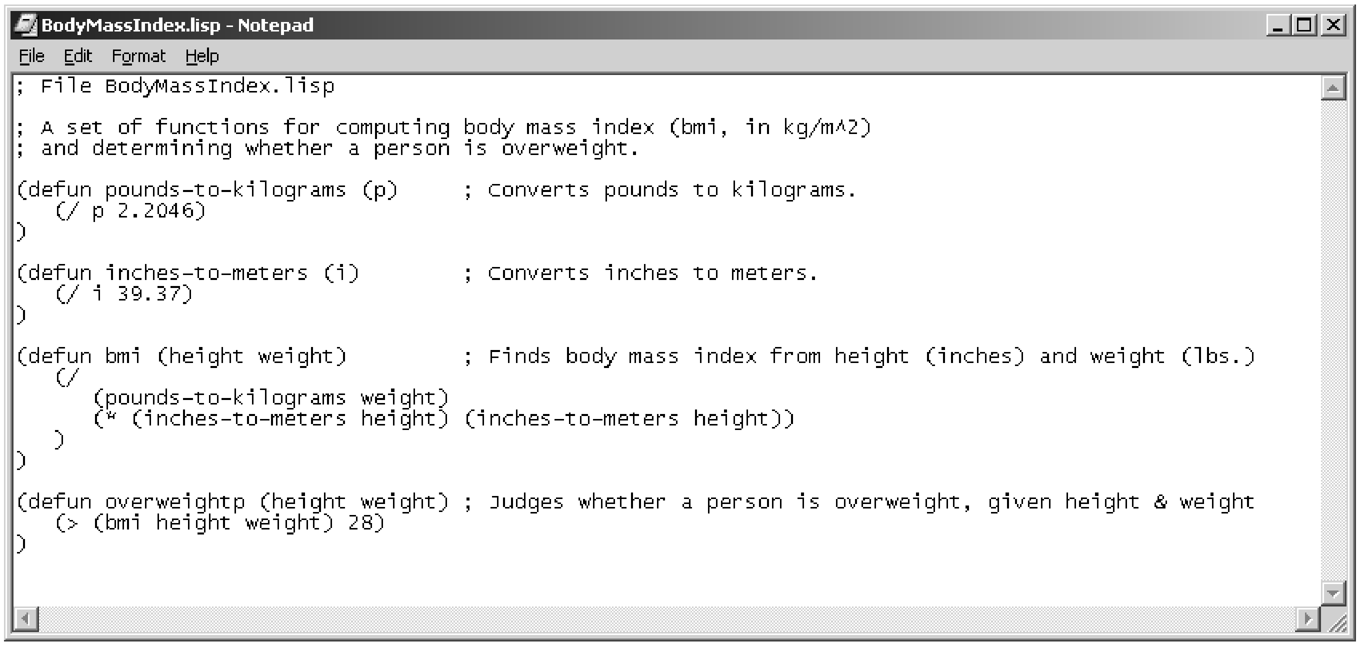
`(if (> 3 2) 'yes 'no)` ⇒ `yes`
`(if (> 2 3) 'yes 'no)` ⇒ `no`
`(if (> 2 3) 'yes)` ⇒ `nil`

A more complex example (assuming `x` has been bound to a value):

```
(if (> x 0)
    'positive
    (if (< x 0)
        'negative
        'zero
    )
)
```

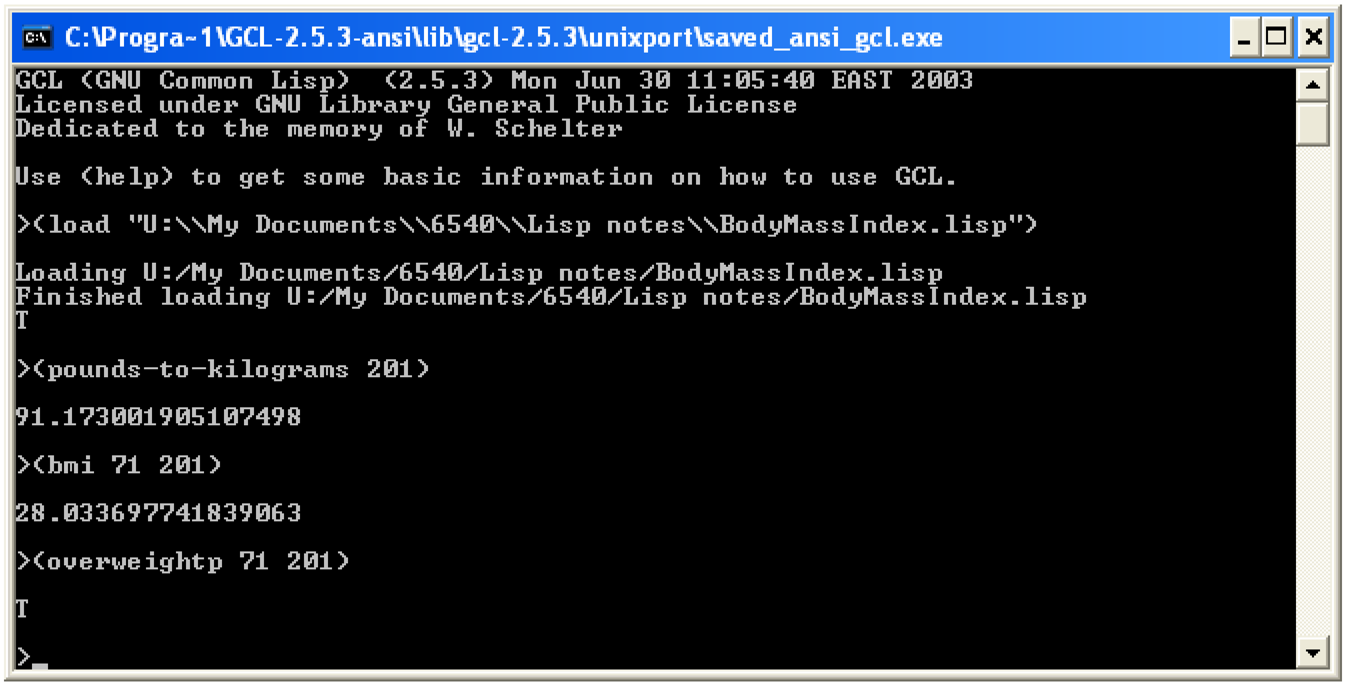
Example of a Lisp program

First create your program on a text file. If using Notepad, make sure to save as ANSI or ASCII, not Unicode.



```
; File BodyMassIndex.lisp

; A set of functions for computing body mass index (bmi, in kg/m^2)
; and determining whether a person is overweight.

(defun pounds-to-kilograms (p)     ; Converts pounds to kilograms.
   (/ p 2.2046)
)

(defun inches-to-meters (i)        ; Converts inches to meters.
   (/ i 39.37)
)

(defun bmi (height weight)         ; Finds body mass index from height (inches) and weight (lbs.)
   (/
      (pounds-to-kilograms weight)
      (* (inches-to-meters height) (inches-to-meters height))
   )
)

(defun overweightp (height weight) ; Judges whether a person is overweight, given height & weight
   (> (bmi height weight) 28)
)
```

Then get into your favorite Lisp system and use `load` to read the program, and then the functions are yours to use.


C:\Progra~1\GCL-2.5.3-ansi\lib\gcl-2.5.3\unixport\saved_ansi_gcl.exe

```
GCL (GNU Common Lisp)  (2.5.3) Mon Jun 30 11:05:40 EAST 2003
Licensed under GNU Library General Public License
Dedicated to the memory of W. Schelter

Use (help) to get some basic information on how to use GCL.

>(load "U:\\My Documents\\6540\\Lisp notes\\BodyMassIndex.lisp")

Loading U:/My Documents/6540/Lisp notes/BodyMassIndex.lisp
Finished loading U:/My Documents/6540/Lisp notes/BodyMassIndex.lisp
T

>(pounds-to-kilograms 201)

91.173001905107498

>(bmi 71 201)

28.033697741839063

>(overweightp 71 201)

T

>_
```

Important: If GNU Common Lisp gives you an error message, you must type `:r` to get back to a normal Lisp prompt.